Information Sciences & Technology

Ames Research Center

Explaining the Cause of an Error

Code with Transient Error

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 tail=(tail+1)%size;
 return buffer[tail];
}
```

Hard to Show Error

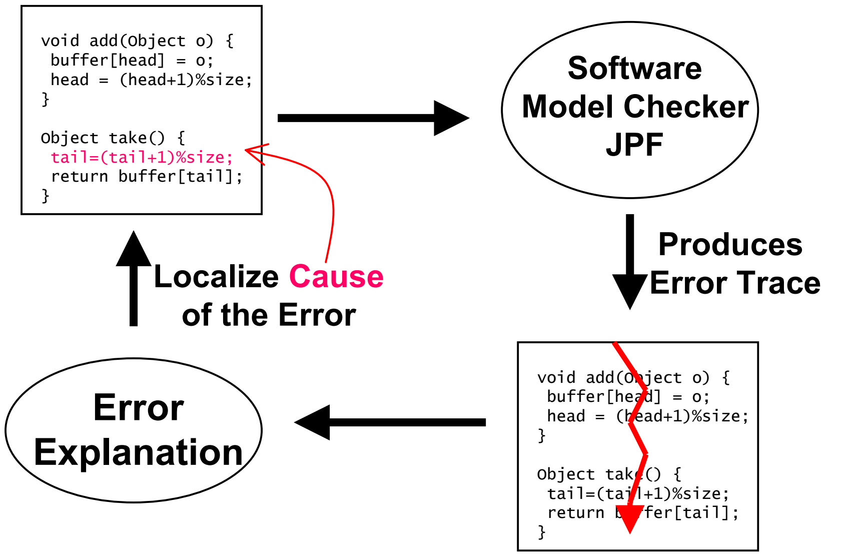
Testing cannot reliably show the error appearing, since it may require specific environment actions (inputs) or scheduling (for concurrency errors)

+

Hard to Find Cause of the Error

Once we know a way to show the error it is difficult to localize the root cause of the error

A model checker can automatically find a trace that shows the error appearing



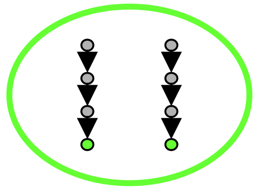
```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 tail=(tail+1)%size;
 return buffer[tail];
}
```

Now we can automatically find an explanation for the error from the error trace produced by the model checker and the original program

The algorithm uses model checking to first find *similar* traces that also cause the error (negatives) and traces that do not cause the error (positives)

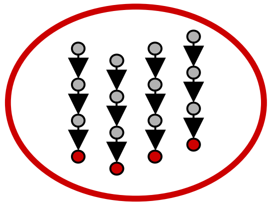
Set of Positives

Traces that don't show the error



Set of Negatives

Traces that show different versions of the error



Analysis

1. **Source code similarities to explain control errors**
   - **code that appear only in negatives**
   - **all negatives, and,**
   - ***only and all* negatives (causal)**
2. **Data invariants – explains errors in data**
3. **Minimal transformations to create a negative from a positive – show the essence of an error**

Explanation of Accomplishment

- **POC:** Willem Visser (ASE group, Code IC, wvisser@email.arc.nasa.gov)
- **Shown:** A transient error is first, hard to make appear, and second, once a symptomatic behavior for the error is found, hard to find the cause of the error. In previous work we have shown that a model checker can be used to automatically find transient errors, such as those caused by concurrency related problems. However, a model checker only produces an error trace and does not give any guidance of what the cause of the error is; here we address this second problem by introducing a technique whereby the cause of an error is explained. The algorithm analyzes two sets of traces in order to explain the cause of an error, namely, a set of traces that show different versions of the error (called negatives) and a set of traces that do not show the error (called positives). From these sets the algorithm either pinpoints a line of code that is most likely to be the cause of the error, a data invariant common to all errors, or a way to take a positive and make a minimal change to it to create a negative.
- **Accomplishment:** The error explanation algorithm have been implemented in the Java PathFinder model checker toolset. It has been used to explain errors in a number of complex software systems, including the DEOS real-time operating system and the Mars K9 Executive prototype. The analysis is confined to behaviors "close" to the original error trace and therefore scales very well (even better than model checking itself). Note, the algorithm is a "in-time" algorithm, i.e. the more time it is given the more accurate are the error explanations.
- **Future Plans:** Our plans are to improve the methods of analysis both to provide more useful feedback and to do more automatic classification of errors (note currently we only classify concurrency errors automatically, but it is possible to classify errors that can only be caused by the environment of the program, etc.). Another possibility is to generate from the negatives an automaton for an environment that avoids reproducing the error – this environment can then be used during runtime monitoring to prevent errors in the real system.